Automatic Transformation from SOFL Module Specifications to Program Structures

Xiongwen Luo(✉) and Shaoying Liu(✉)

Department of Computer and Information Sciences, Hosei University, Tokyo, Japan
Xiongwen.luo.2b@stu.hosei.ac.jp, sliu@hosei.ac.jp

Abstract. The Structured Object-oriented Formal Language (SOFL) method is developed to overcome the disadvantages of existing formal methods and provide effective techniques for writing formal specifications and carrying out verification and testing. Although it has been applied to system modeling and design in practical and research projects, SOFL has not been widely applied to the industrial software development systems because of the lack of efficient tool support. Aiming at improving the existing SOFL supporting tool and solving the problem that the formal specifications cannot be directly executed, this paper firstly analyzes the relationship between the structures of SOFL formal specifications and C# programs, and then discusses how module transformation and data type transformations are implemented. Finally, a testing is performed to ensure the reliability of the implemented software system.

Keywords: SOFL · Specification transformations · Data type · Programs

1 Introduction

Specification-based testing and inspection for programs are two major techniques in the SOFL method [1], but they are facing challenges due to possible inconsistency between SOFL formal specifications and programs in some programming language such as Java or C#. The inconsistency may exist in process signatures, data types, or the structure of the documents. A process in SOFL is an operation transforming input to output, but with multiple input and/or output ports, which differs from an operation in other formal notations, such as VDM [2], Z [3], and B-Method [3]. When a process is implemented in a program, it is usually realized by a method in a class. It is highly possible that the parameters of the method are inconsistent with those of the process in the specification. Further, the data types adopted in SOFL, such as set, sequence, map, composite types, may be easily implemented using similar data types in the programming language, but the associated operators defined on those types may be represented using the different syntax in both the specification and the program. Thus, test cases generated based on the specification may not be directly applicable in executing the program for testing. This sets a big hurdle for a complete automatic testing technique to be established.

To deal with this problem, in this paper we describe a new approach called *signature-preserved transformation*. The essential points of the approach are threefold.

DOI: 10.1007/978-3-319-31220-0_15

Firstly, the signature of each process in the specification is automatically transformed into a method signature that preserves the number of the parameters and their types declared in the process specification. The body of the method is left to the programmer to complete manually based on the pre- and post-conditions of the process specification. Secondly, all of the SOFL data types and all the associated operators are implemented in the programming language in the manner their syntax is almost preserved. Thus, once the package containing the implementation classes of the data types is imported, the same operators with the same syntax in SOFL can be directly used in the program. This will facilitate the application of test cases generated from the specification to the program implementing the specification. Finally, the structure of the entire module structure can be automatically transformed into a class structure in which all of the related constants, type declarations, state variable declarations are properly presented.

The rest of this paper is organized as follows. Section 2 briefly describes the structure of module in SOFL. Section 3 analyzes the principles of transformation. These transformations are process transformation, module structure transformation, and data type transformation. Section 4 discusses the design and implementation of transformation. Section 5 presents the testing of the programming for verifying the validation and reliability of the programs. The related work is given in Sect. 6. Finally, the last section, Sect. 7, concludes the work of this paper, and points out the future research directions.

2 The SOFL Module

In this section, we briefly introduce the structure of a SOFL module in order to help the reader understand our discussions on automatic transformation late in the paper.

A module in SOFL is a textual document defining the semantics of all of the components occurring in an associated condition data flow diagram (CDFD) [3]. The most important part of the module is the process specifications. Each process models a transformation from input data flows to output data flows and its functionality is specified

module ModuleName
const ConstantDeclaration;
type TypeDeclaration;
var VariableDeclaration;
inv TypeandStateInvariants;
behav CDFD_no;
InitializationProcess;
process_1;
process_2;
    …
function_1;
function_2;
    …
function_m;
end_module

Fig. 1. Structure of module

using pre- and post-conditions. All of the data flows are associated with processes in the way they are used as either input or output. Each process can also access or update some data stores where each data store is represented by a variable of some type. For the purpose of constructing the process specifications, necessary constants identifiers and type identifiers may need to be declared, and all of the data store variables must be declared with proper types in the module, as illustrated in Fig. 1. Another important part is function definitions. Functions can be applied in the pre- or post-conditions of some process specifications, but for this purpose, they must be defined either explicitly or implicitly. We omit the details here for the sake of space. The reader who is interested in the details can refer to the SOFL book [3].

3 Principles of Transformation

In this section, we focus on the principles for process transformation, module struc-ture transformation, and data type transformation. From the next section, we will extend our discussion to the implementation of the transformation principles.

3.1 Process Transformation

A process, basically, consists of five parts: process name, input data flow variables, output data flow variables, pre-condition and post-condition. The process presents an action or operation that consumes the input data flows and generates the output data flows. If there are external variables that need to be used in this process, they are stated after the keyword **ext**. A complex process may be decomposed into the lower level CDFD whose associated module is written after the keyword **decom**. The keyword **comment** starts the informal comment section, which is usually written to improve the readability of the formal specifications (Fig. 2).

process ProcessName(input) output
ext ExternalVariable
pre PreCondition
post PostCondition
decom LowerLevelModuleName
explicit ExplicitSpecification
comment
end_process

Fig. 2. The structure of process

As process is one of the most essential parts in module [4], we firstly transform the process to the programs. The process has two types: one is the single-port process with only one input port and one output port, the other is the multiple-port process with exclusive input or output data flows. When dealing with the transformations of process, we implement these two types in two different forms.

```
process A(x_1: Ti_1, x_2: Ti_2, ..., x_n: Ti_n)
        y_1: To_1, y_2: To_2, ..., y_m:To_m
pre pre_A
post post_A
end_process
```

Fig. 3. General form of single-port process

```
class ModuleName{
To_1 y_1;
To_2 y_2;
…
To_m y_m;
public void A(Ti_1 x_1, Ti_2 x_2, …, Ti_n x_n){
      if(pre_A) Tran(post_A)
  …
   }
}
```

Fig. 4. Transformation of single-port process

Firstly, the general form of single-port process is presented in the Fig. 3, and transforming this process into method is shown in Fig. 4.

Then, the general form of multiple-port process is presented in the Fig. 5, and transforming this process into method is shown in Fig. 6.

```
process A(x_1: Ti_1| x_2: Ti_2 | ... | x_n: Ti_n)
        y_1: To_1 | y_2: To_2 | ... | y_m:To_m
pre pre_A
post post_A
end_process
```

Fig. 5. General form of multiple-port process

```
class ModuleName{
To_1 y_1;
To_2 y_2;
…
To_m y_m
public To_1 A_ y_1(Ti_1 x_1, Ti_2 x_2, …, Ti_n x_n){
      if(pre_A(x_1))
    {
      Tran(post_A)
    }
  }
public To_2 A_ y_2(Ti_1 x_1, Ti_2 x_2, …, Ti_n x_n){
      if(pre_A(x_2))
    {
      Tran(post_A)
    }
  }
…
public To_m A_ y_m(Ti_1 x_1, Ti_2 x_2, …, Ti_n x_n){
      if(pre_A(x_n))
    {
      Tran(post_A)
    }
  }
}
```

Fig. 6. Transformation of multiple-port process

3.2 Module Transformation

As illustrated in Fig. 1, the beginning of the module is the keyword **module**. *Module-Name* is a unique identifier of module in SOFL specifications. The key words **const**,

type and **var** start the sections for constant declarations, type declarations, and variable declarations respectively. The key word **inv** stands for the type and state invariants, which represents the constraints on the type declarations section and variable declarations section. The *CDFD_no* after the key word **behav** specifies the affiliated CDFD. The last two parts, beginning with keyword **process** and **function**, offers some processes and functions.

Apparently, a module is similar to a class in C# in structure. Therefore, we take the straightforward principle to transform a module into a C# class. Specifically, the ideas of transforming each part of the module are given as follows:

- Transform the constant declaration to the constant in C#, using the keyword **const** prior to the constant variables.
- Transform the type declaration to either a basic type or a class, whose form is in compliance with C# language syntax. However, the data type is written in SOFL language, so that it needs to be implemented by C# language as I discuss later.
- Transform the variable declarations to the instance variables, stored and accessed in the external file, but used in this transformation class.
- Transform the processes to the target methods.
- Transform the functions to the target methods, which are similar to that of processes.

3.3 Data Type Transformation

Data types, an essential part for specifications, provide a notation to define data structures in the SOFL formal specification [5]. Although we can transform a module to a class in the programming language, only completing data type transformations can the results of module transformations be used for final automatic specifications testing. Because the data types in SOFL are not identical with C# language data types both in semantics and syntax [6], we cannot directly execute the results of module transformations. In other words, it attaches no significance to the transformations of module without the data type transformations. Only with the support of data types transformations, results of module transformations can be used for specifications and program testing.

In SOFL language, the data types are divided into two categories: built-in type and user-defined types. The built-in types have fourteen kinds of data types, which are further divided into basic types and compound types. The basic types include *nat0* type, *nat* type, *int* type, *real* type, *bool* type, *char* type, *string* type and *Enumeration* type, while the compound types are: *set* type, *sequence* type, *composite* type, *map* type, *product* type and *union* type. The user-defined types are defined by the specification writers. They are based on the built-in data types, so the transformation guidelines of built-in data types also apply to the transformation of user-defined types.

It is worth noting that transformations from data types in SOFL to the data types in C# language require both semantics preservation and syntactic changes [7]. Firstly, the syntax of variable declaration in SOFL language is not identical with that of variable declaration in C#. The former lets the type appear behind the variable with a colon separating them, while the latter makes the type appear prior to the variable with a space between them. What is more, in the semantics perspective, some of the types are in accordance with that in C#, such as *int* type, *char* type, *string* type, *bool* type and *Enumeration* type. While some of the types are similar to that in C#, for example *nat0* type, *nat* type and *real* type. In addition, some of the types are different from that in C#, for instance, *set* type, *sequence* type, *composite* type, *map* type, *product* type and *union* type.

In general, the choice of the concrete data types in the transformation will affect somehow the algorithms of the implemented program using the data types [8, 9]. Therefore, it is essential to strike a balance between data structures and algorithms. Under this circumstance, we consider that some of the data types do not need to be transformed, as they have already existed in the C# and can be executed directly. While some of the data types need to be implemented by the similar data types available in C#, the transformations of which can be quite straightforward owing to the support of.Net platform with rich data structures and class libraries.

Based on this analysis, we adopt several principles for data type transformation summarized as follows:

- The *int* type, *char* type, *string* type, *bool* type, and *Enumeration* type do not need to be transformed because they have already existed in C# and can be used directly.
- The *nat0* type defines the natural numbers including zero and *nat* type defines the natural numbers, so that these two types can be implemented by *int* type in C#.
- The *real* type represents the real numbers, which can be implemented by *double* type in C#.
- The *set* type is an unordered collection of distinct objects, which is similar to the *HashSet* type in C#, so it is natural to implement it through *HashSet* type.
- The *sequence* type is an ordered collection of objects that allows duplications of objects. Taking this into account, I believe that *List* type in C# is the best choice to implement it.
- The *map* type is a finite set of pairs, the domain and range to the map share the similar meaning with the key and value to the *dictionary* type in C#.
- The *composite* type and *product* type represent a collection of several data items, so the abstract classes are used to implement these two types and their inherent functions.
- The *union* type is a special type associated with several functions. It can be regarded as a collection of variables in different types. We consider that we will transform this type to a class with many fields in C#.

4 Design and Implementation of Transformations

In general, we create three packages to implement the transformations. One is automatic transformation package, and other two are module transformations package and data type transformations package as shown in Fig. 7. In the automatic transformation package, we need to invoke the methods defined in the module transformations package to complete the module transformations. The results of transformations cannot be executed without the support of data type transformations which are completed in data type transformations package.

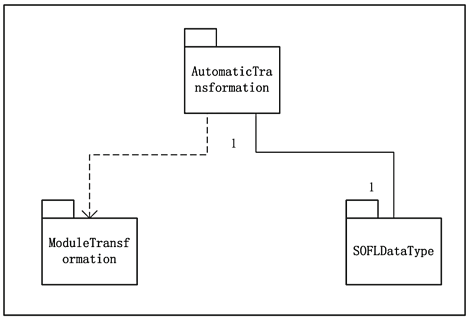


Fig. 7. The structure of transformation framework

The AutomaticTransformation package includes the main class of the transformations, whose *main* method is the entry of the automatic transformations. This package invokes the methods of classes in the ModuleTransformation package and is supported by the classes in the SOFLDataType package.

4.1 Main Program in AutomaticTransformation

The AutomaticTransformation package includes the main program used to complete the automatic transformations. The *main* method in this program is the entry of the automatic transformations and it invokes the methods of classes in the ModuleTransformation package.

In this process, firstly, we enter the path of XML file, and then judge whether the file exists or not. If the XML file exists, we have to enter the output file path and also make a judgement to ensure that the file name is legal. In the next step, we will make a choice to decide whether to start the transformations or not. If we choose to start the transformations, the specifications will be transformed to C# programs. After completing the transformation, the system will terminate.

4.2 Implementation Classes in ModuleTransformation

Figure 8 shows the classes in ModuleTransformation, the details of each class is introduced in the following:

- XmlTool class: The objective of this class is to provide a XML file tool used for extracting the data information of SOFL formal specifications form the XML files. Before executing the transformations process, we should make formal specifications generate the corresponding XML files through the existing SOFL supporting tool, then this class is to parse these XML files to get the data information we need.
- ConstantTransformation class: This class mainly deals with constant declaration in SOFL formal specifications. It contains two methods, one is to get the constant variables and write into the external file, and the other is to judge constant variables types and invoke the former method to write the corresponding constants.
- TypeTransformation class: This class is used to complete the type declaration transformations. Because there are many kinds of different data types, we need to invoke

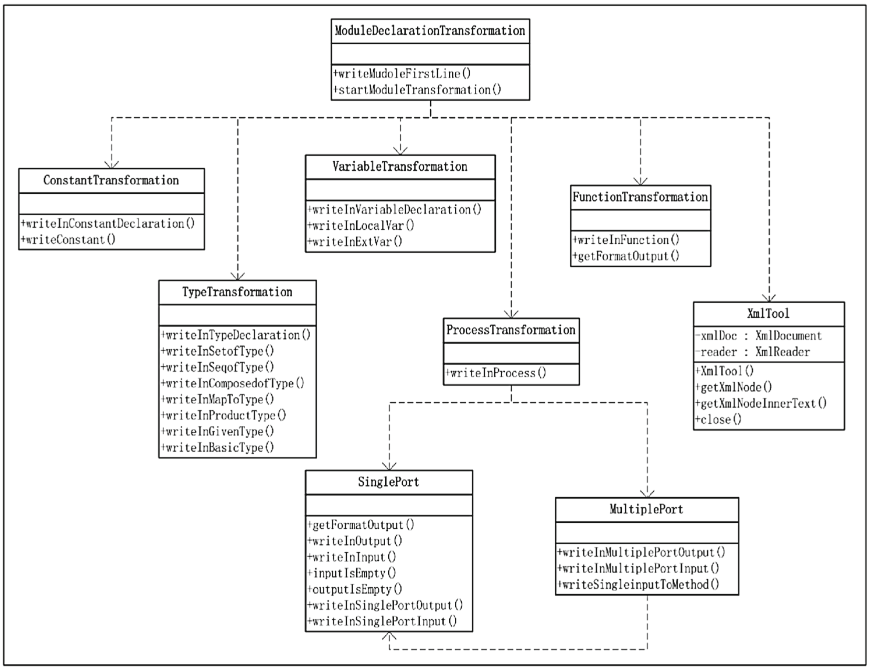


Fig. 8. The implementation classes in ModuleTransformation

different methods to implement the transformations. In other words, each of the compound type transformation uses one method.

- VariableTransformation class: The purpose for this class is to transform the variable declaration section to the programing in C#. Since the variables appearing in this corresponding part are either the local variables or external variables, so we design two methods to implement this process. One is for writing the local variables into target files and the other is for writing the external variables.
- FunctionTransformation class: This class is to handle the transformations from function declaration to programs. Owing to the similar structure with the method in C#, we design a method to write these function declarations into the target files.
- ProcessTransformation class: In this class, a method is defined to judge different cases, and for each case, we invoke the methods of SinglePort class to execute the single-port process transformations and the methods of MultiplePort class to execute the multiple-port process transformations.
- ModuleDeclarationTransformation class: This class is to realize the functions of module transformations by invoking the methods in other classes. It has two methods, one is to write the first line of module and the other is to complete the transformations of other parts in module sequentially.

4.3 Implementation Classes in SOFLDataType

There are fourteen kinds of data types in SOFL, but five of them share the same semantics and syntactic with C# language so that they can be directly executed in the programming. The rests need to be transformed in the C# to support the results of module transformations. In this case, we design nine interfaces to implement the transformations of nine kinds of data types. The relationship is shown in Table 1.

Table 1. Data types in SOFL and their implementation classes

Key word of data type in SOFL	Data type interface	Implementation class
nat0	Inat0	nat0
nat	Inat	nat
real	Ireal	real
set	Iset	set
seq	Iseq	seq
map	Imap	map
composite	Icomposite	composite
product	Iproduct	product
union	Iunion	union

Note that the naming conventions of class in C# is that the first letter of class name is capitalized, but we do not observe this rule because we want to make the name of implementation classes in accordance with the keyword of data type in SOFL, so as to use the implementation classes efficiently and unambiguously. The more details about the design of data type interfaces and the implementations of methods in corresponding classes are presented by the UML class diagram as follows:

We design nine interfaces, which are *Inat0*, *Inat*, *Ireal*, *Iset*, *Iseq*, *Imap*, *Icomposite*, *Iproduct* and *Iunion*, to implement the transformations of these nine kinds of data types. The methods in each interface are consistent with the operators in the related SOFL data types. Then, the nine classes, which are nat0, nat, real, set<T>, seq<T>, map<T, E>, composite, product and union, are created to implement the corresponding interfaces (Fig. 9).

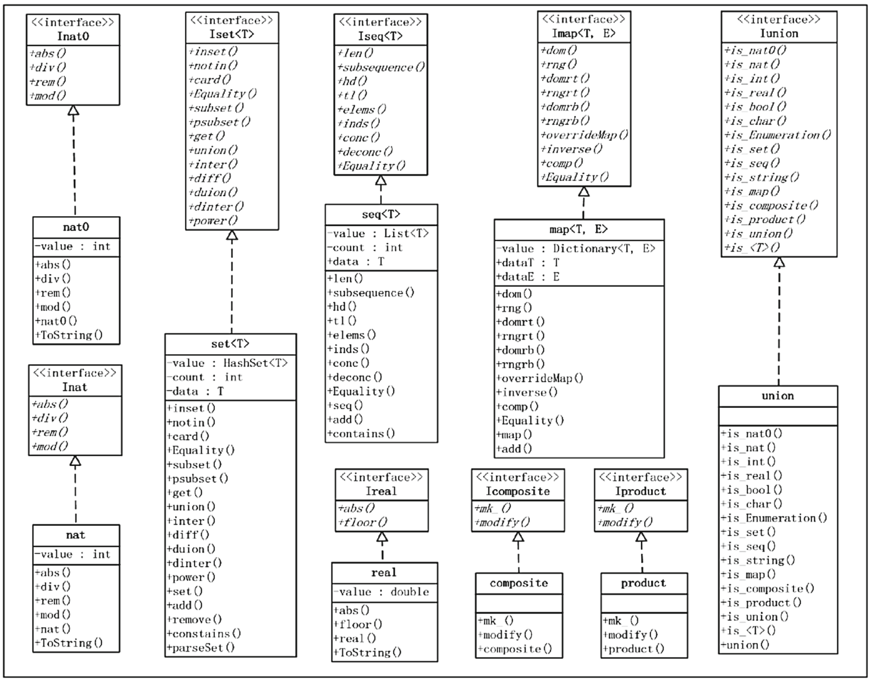


Fig. 9. The implementation classes in SOFLDataType

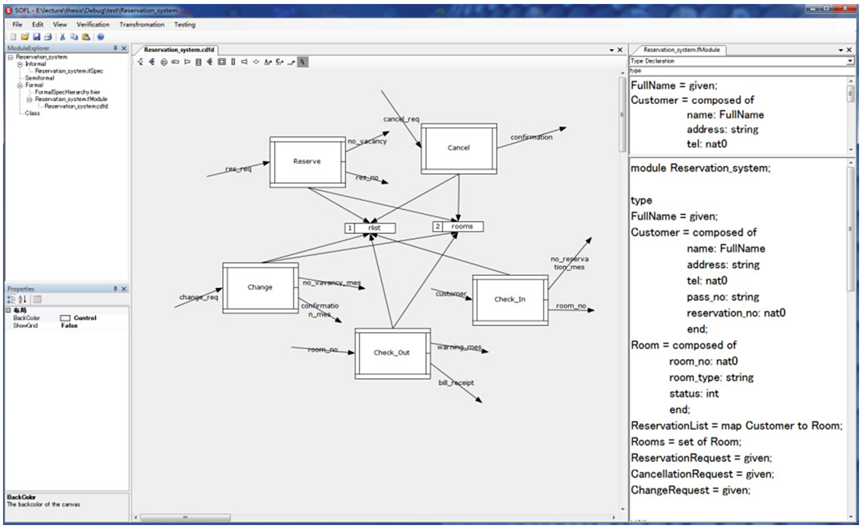
5 Transformation Results

After the transformation software system is implemented, it is essential to perform a test to detect faults and ensure the validity and robustness of the system. In order to check whether each function in the transformations can be used correctly, we adopt the black-box testing method to test the transformations process. Firstly, unit testing method is used to test each transformation section, and then the integration testing methods are adopted to ensure the success of the entire transformations (Fig. 10).

The testing procedures can also be the guidance of how to use these programs to make the automatic transformation, which are listed as follows:

1. Using the existing SOFL supporting tool to create the formal specifications and draw the related CDFDs [10].

   In the existing SOFL supporting tool, we can use the three-step approach to constructing the formal specifications. The structure of the components in current project is displayed in the upper-left corner. In the center, a CDFD related to the module is drawn. If one item in the CDFD is selected, the attributes of it will be presented in the lower- left corner. The module in detail is written in the right side.
2. Generating the related XML file through the existing SOFL supporting tool.



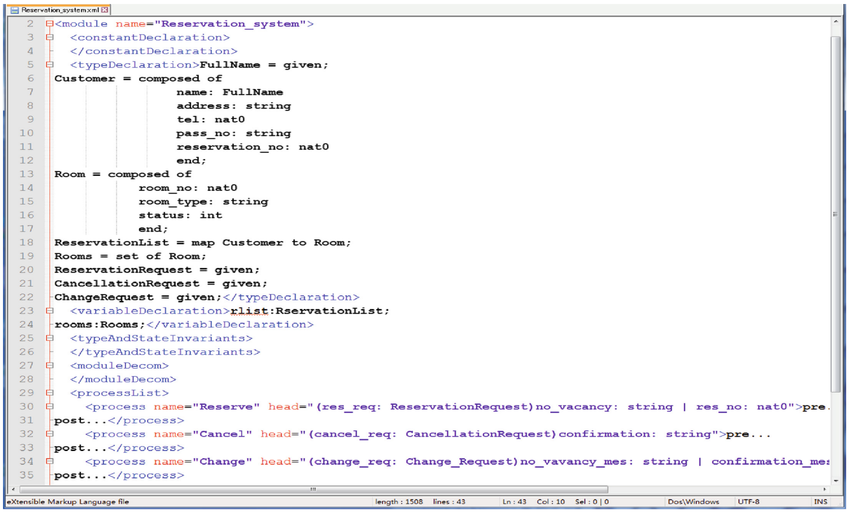
```
FullName = given;
Customer = composed of
        name: FullName
        address: string
        tel: nat0
```

```
module Reservation_system;

type
FullName = given;
Customer = composed of
        name: FullName
        address: string
        tel: nat0
        pass_no: string
        reservation_no: nat0
        end;
Room = composed of
      room_no: nat0
      room_type: string
      status: int
      end;
ReservationList = map Customer to Room;
Rooms = set of Room;
ReservationRequest = given;
CancellationRequest = given;
ChangeRequest = given;
```

Fig. 10. Snapshot of the SOFL formal specification of a hotel reservation system

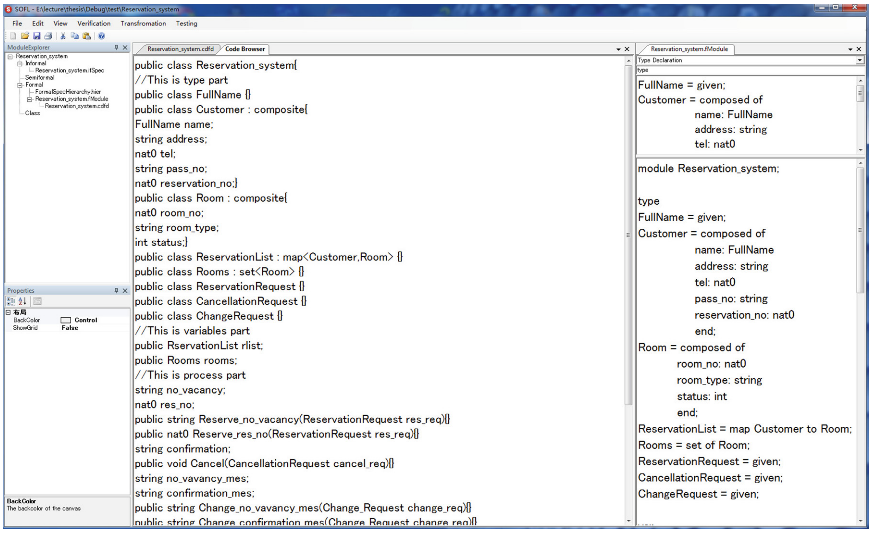
In Fig. 11, the names of labels are related to the corresponding keywords in the SOFL formal specifications constructed in step 1. For example, the label "module" is related to the keyword **module** in the specification.



```
<module name="Reservation_system">
  <constantDeclaration>
  </constantDeclaration>
  <typeDeclaration>FullName = given;
Customer = composed of
                name: FullName
                address: string
                tel: nat0
                pass_no: string
                reservation_no: nat0
                end;
Room = composed of
          room_no: nat0
          room_type: string
          status: int
          end;
ReservationList = map Customer to Room;
Rooms = set of Room;
ReservationRequest = given;
CancellationRequest = given;
ChangeRequest = given;</typeDeclaration>
  <variableDeclaration>rlist:RservationList;
rooms:Rooms;</variableDeclaration>
  <typeAndStateInvariants>
  </typeAndStateInvariants>
  <moduleDecom>
  </moduleDecom>
  <processList>
    <process name="Reserve" head="(res_req: ReservationRequest)no_vacancy: string | res_no: nat0">pre.
post...</process>
    <process name="Cancel" head="(cancel_req: CancellationRequest)confirmation: string">pre...
post...</process>
    <process name="Change" head="(change_req: Change_Request)no_vavancy_mes: string | confirmation_me
post...</process>
```

Fig. 11. XML file of the related SOFL formal specification

3. Using the software system we have developed to parse the XML file and complete the transformation. A result of the transformation is presented in Fig. 12.



```
public class Reservation_system{
//This is type part
public class FullName {}
public class Customer : composite{
FullName name;
string address;
nat0 tel;
string pass_no;
nat0 reservation_no;}
public class Room : composite{
nat0 room_no;
string room_type;
int status;}
public class ReservationList : map<Customer,Room> {}
public class Rooms : set<Room> {}
public class ReservationRequest {}
public class CancellationRequest {}
public class ChangeRequest {}
//This is variables part
public RservationList rlist;
public Rooms rooms;
//This is process part
string no_vacancy;
nat0 res_no;
public string Reserve_no_vacancy(ReservationRequest res_req){}
public nat0 Reserve_res_no(ReservationRequest res_req){}
string confirmation;
public void Cancel(CancellationRequest cancel_req){}
string no_vavancy_mes;
string confirmation_mes;
public string Change_no_vavancy_mes(Change_Request change_req){}
public string Change_confirmation_mes(Change_Request change_req){}
```

```
FullName = given;
Customer = composed of
            name: FullName
            address: string
            tel: nat0
```

```
module Reservation_system;

type
FullName = given;
Customer = composed of
            name: FullName
            address: string
            tel: nat0
            pass_no: string
            reservation_no: nat0
            end;
Room = composed of
        room_no: nat0
        room_type: string
        status: int
        end;
ReservationList = map Customer to Room;
Rooms = set of Room;
ReservationRequest = given;
CancellationRequest = given;
ChangeRequest = given;
```

Fig. 12. Results of transformations

In Fig. 12, the module name is related to the class name. The constant declarations in the module are transformed to the constant variables. The type declarations are transformed to either a basic type or a class. The variable declarations are transformed to the instance variables. The process and function are implemented by the target methods.

6 Related Work

There exist some tools to support automatic transformation from other formal notation to programming languages. VDMTools [11] offer the functions of analyzing the system models expressed in the formal language of VDM, which has been applied to developing industrial software systems. The VDM specification can be executed directly through the interpreter inside this tool. ProB [12] is a validation toolset for the B method. In this tool, a model checker and a refinement checker can be used for executing the B specifications to detect various errors. UPPAAL [13] is a verification tool for timed automate. In UPPAAL, there is no textual specification, but user can construct the finite state machine to module the functions of a system, which can be executed in this tool to detect faults.

7 Conclusions and Future Work

In this paper, we discuss the principles and implementation of automatic transformations from SOFL formal specifications to programs. We first analyzed the module structure and data type in SOFL formal specifications, which lays the foundation for the transformations. We then described the design and implementation of transformations. After completing the transformations, we used a black-box testing method to design test cases and carry out unit testing, integration testing and system testing methods to verify the results of transformations.

In the future, we will continue this transformations work and plan to extend to the CDFD and class in SOFL formal specifications. With the development of automatic transformations, we would be interesting in the applications of these transformations for specifications testing and automatic generation of test cases.

Acknowledgment. This work was supported by JSPS KAKENHI Grant Number 26240008.

References

1. Liu, S., Chen, Y., Nagoya, F., McDermid, J.: Formal specification-based inspection for verification of programs. IEEE Trans. Softw. Eng. **38**(5), 1100–1122 (2012)
2. Mosses, P.D.: VDM semantics of programming language: combinators and monads. Formal Aspects Comput. **23**, 221–238 (2011)
3. Liu, S.: Formal Engineering for Industrial Software Development: Using the SOFL Method. Springer, Heidelberg (2004). ISBN 3-540-20602-7
4. Liu, S.: An approach to applying SOFL for agile process and its application in developing a test support tool. Innovations Syst. Softw. Eng. **6**(1), 137–143 (2009). Springer
5. Zainuddin, F.B., Liu, S.: An approach to low-fidelity prototyping based on SOFL informal specification. In: IEEE APSEC (2012), ISSN: 1530-1362/12
6. Miao, W., Liu, S.: Service-oriented modeling using the SOFL formal engineering method. In: IEEE APSCC (2009), ISBN: 978-1-4244-5336-8/09
7. Liu, S., Xue, X.: Automated software specification and design using the SOFL formal engineering method. In: IEEE WCSE (2009), ISBN: 978-0-7685-3570-8/09
8. Chen, Y., Liu, S., Nagoya, F.: A Framework for SOFL-based Program Review. In: IEEE ICECCS (2005), ISBN: 0-7695-2284-X/05
9. Chen, Y.: A case study of using SOFL to specify a concurrent software system. In: IEEE (2010), ISBN: 978-1-4244-6055-7/10
10. Li, M., Liu, S.: Tool support for rigorous formal specification inspection. In: IEEE CSE 2014 (2014), ISBN: 978-1-4799-7981-3/14
11. Fitzgerald, J., Larsen, P.G., Sahare, S.: VDMTools: advances in support for formal modeling in VDM. ACM SIGPLAN Not. **43**(2), 3–11 (2008)
12. Leuschel, M., Butler, M.: PROB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**, 185–203 (2008)
13. Behrmann, G., David, A., Larsen, K.G.: UPPAAL 4.0. In: IEEE QEST06 (2006), ISBN: 0-7695-2665-9/06